ISSN 1512-1232

Code 004.9

Building a Symmetric Cryptosystem Using Genetic Algorithms and PRNG

Beselia Lali

Faculty of Mathematics and Computer Sciences, Sokhumi State University, Tbilisi, Georgia

l.beselia@sou.edu.ge

Abstract

In today's information-oriented world, ensuring secure and efficient communication is of paramount importance. Traditionally, the security of symmetric cryptosystems relies on the complexity of mathematical calculations. However, modern approaches have begun to explore alternative methods that combine mathematical calculations with innovative methods. In this paper, we present a symmetric cryptosystem built using genetic algorithms and a pseudorandom number generator. The article provides a description of the cryptosystem construction algorithm, its implementation in the Python programming language, and an analysis of the cryptosystem.

Introduction

A cryptosystem that combines two elements in the encryption process:

1. Genetic operations: By using genetic operations such as crossover (where parts of binary data are swapped) and mutation (where individual bits are inverted), the system increases the randomness and complexity of the ciphertext.
2. PRNG: The use of a PRNG to determine the crossover and mutation points ensures unpredictability and speed of the encryption process [1].

Description of the algorithm

1. In the first stage, the plaintext is converted to binary and divided into blocks. Each character of the given text is represented by its ASCII code, whichis then converted to a binary number.

```
def text_to_binary(text):
binary_data = ''.join(format(ord(char), '08b') for char in text)
return binary_data
```

The resulting binary string is divided into 128-bit blocks. If the last block is not large enough, it is padded with 0-s.

```
def split_into_blocks(binary_data, block_size=128):
blocks = [binary_data[i:i+block_size] for i in range(0,
            len(binary_data),block_size)]
if len(blocks[-1]) < block_size:
blocks[-1] = blocks[-1].ljust(block_size, '0')
return blocks
```

ISSN 1512-1232

Dividing into blocks of equal size is crucial for performing the next stages of encryption.

1. The second stage is key generation. Creating a secret key is an important component of the encryption process. In our system, a 128-bit key is created by generating 16 random non-zero integers (each ranging from 1 to 255). The integers are then converted to binary format, after which the key is used to perform an xor operation on the text blocks. The following program code fragment is responsible for key generation:

```
def generate_key():
    key = []
    while len(key) < 16:
        num = secrets.randbelow(256)   # Generates a number
between 0 and 255.
        if num != 0:
            key.append(num)
    return key

def key_to_binary(key):
    binary_key = ''.join(format(num, '08b') for num in key)
    return binary_key
```

2. After generating the secret key and converting it to binary, a bitwise XOR operation is performed between each 128-bit block of plaintext and the key.

```
def xor_blocks_with_key(blocks, key):
    encrypted_blocks = []
    for block in blocks:
        encrypted_block = ''.join(str(int(b1) ^ int(b2)) for
b1, b2 in zip(block, key))
        encrypted_blocks.append(encrypted_block)
    return encrypted_blocks
```

The given method processes each plaintext block in sequence, ensuring that the key is applied to each bit of each block. As a result, we obtain an intermediate ciphertext, which is further disguised using genetic operations.

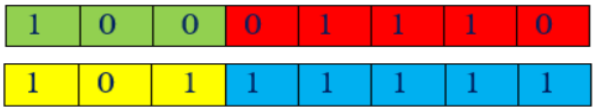
3. The crossover operation is performed. Using the PRNG, we calculate the first sixteen pseudorandom numbers. The resulting 16 pseudorandom numbers represent the crossover (crossover) points of the genetic operation between bytes. Similar to the secret key, we exclude zero values here. At these points, the crossover is performed according to the following principle: the first and second bytes of the first block are crossed at the “0” point, then the second and third are crossed at the “1” point, and so on until the sixteenth byte is crossed with the first byte of the block [4].def apply_crossover(blocks, crossover_points):

```
  new_blocks = []
    for block in blocks:
        bytes_list = [block[i:i+8] for i in range(0, len(block),
8)]
        for i in range(len(bytes_list) - 1):
            cp = crossover_points[i]
            bytes_list[i],bytes_list[i+1]=
 crossover(bytes_list[i], bytes_list[i+1], cp)
          new_blocks.append(''.join(bytes_list))
```

ISSN 1512-1232

To make it clearer how the concatenation process is performed, let's give an example. Let's say we have two bytes:

If the crossover point 3 is "3", then the result of the crossover at this point is:



4. In the fifth stage, a mutation operation is performed, which further enhances randomness by changing one bit in each byte at a pseudo-randomly chosen position.[5] The mutation function is implemented as follows: def mutate_byte(byte, point):

```
byte_list = list(byte)
byte_list[point] = '1' if byte_list[point] == '0' else '0'
return ''.join(byte_list)
```

For example, given a byte:

If the mutation operation is performed on the third bit of the byte, we get:

After the five steps above, if each byte of each block is different from the corresponding byte of the corresponding block of plaintext, meaning that it is enough for the plaintext information to be well hidden in the resulting ciphertext, then the bytes are converted to symbols. Otherwise, we repeat several rounds, and during each round the key and intersection points change.

Decryption is performed in the reverse order of encryption operations:

1. Decryption begins by converting the ciphertext to binary. Each symbol is converted into an 8-bit binary string:

```
binary_data = ''.join(format(ord(char), '08b') for char in ciphertext)
```

Once the ciphertext is represented in binary format, it is divided into 128-bit blocks. If the last block is less than 128 bits, it is padded with zeros:

```
def split_into_blocks(binary_data, block_size=128):
blocks = [binary_data[i:i+block_size] for i in range(0, len(binary_data), block_size)]
    if len(blocks[-1]) < block_size:
        blocks[-1] = blocks[-1].ljust(block_size, '0')
    return blocks
```

ISSN 1512-1232

2. During encryption, each byte is changed by changing one specific bit at the mutation point; during decryption, the reverse process is performed, with a specific bit being changed in each byte of the ciphertext.def reverse_mutation(blocks, mutation_points):

```
    new_blocks = []
    for block in blocks:
        bytes_list = [block[i:i+8] for i in range(0, len(block),
8)]
        for i in range(len(bytes_list)):
            mp = mutation_points[i]
            # Flipping the bit again to revert the mutation
            bytes_list[i] = mutate_byte(bytes_list[i], mp)
        new_blocks.append(''.join(bytes_list))
    return new_blocks
```

3. Before mutation, a crossover operation was applied to each 128-bit block. The block was divided into 16 bytes, and adjacent bytes exchanged parts of their bits at a certain crossover point. Decryption reverses this by processing the bytes in reverse order and applying the same crossover operation.def reverse_crossover(blocks, crossover_points):

```
  new_blocks = []
    for block in blocks:
        bytes_list = [block[i:i+8] for i in range(0, len(block),
8)]
        for i in reversed(range(len(bytes_list) - 1)):
            cp = crossover_points[i]
            bytes_list[i],bytes_list[i+1]=
crossover(bytes_list[i], bytes_list[i+1], cp)
        new_blocks.append(''.join(bytes_list))
    return new_blocks
```

This step of the inverse process undoes the interleaving operation in a way that restores the original arrangement of bits in the blocks.

4. The original encryption used an XOR operation between each 128-bit block and a 128-bit key. Because of the symmetry of XOR, reusing it yields the original data:def xor_blocks_with_key(blocks, key):

```
    result = []
    for block in blocks:
        result.append(''.join(str(int(b1) ^ int(b2)) for b1, b2 in
zip(block, key)))
    return result
```

5. Convert to plain text:

Every 8 bits are converted to a symbol:def binary_to_text(binary_data):

ISSN 1512-1232

```
    return ''.join(chr(int(binary_data[i:i+8], 2)) for i in
range(0, len(binary_data), 8))
```

We present the results of the algorithm's work:

```
Original text: This is a secret message.

--- Encryption Round 1 ---

Final ciphertext (hex):
2ad818f49a3b8ed862e58a650bc4d1d64d525ab5b472ff7b1e46e518f479b7e4

Final ciphertext (base64):
KtgY9Jo7jthi5YplC8TR1k1SWrWOcv97HkblGPR5t+Q=

--- Decryption Process ---

Decrypted text:
This is a secret message.
```

Result 1.

```
Original text: Georgia is a small country with a great and ancient history. For centuries, it has
    been a constant battleground for great empires due to its strategic location. However, it still
    managed to survive and maintain its statehood.

--- Encryption Round 1 ---

Final ciphertext (hex):
d772b4ea01e2c7b3827bec2dacd4e809c87cffa06b4c7df9996bfa2c8765d844d395bdba6c6af59b2b35e17dc55eba10c30
    82ed525615df986408b759706cd4bd005907b38f9d8efb655c574c3799030c440f80c79e8eba9a91aba41b649f01dc5
    40dcbc6e765de6c27f8a08cd5cfd5b5c66acca6cef985ebd7cb84dab6afc129732ba9864a6e288ce2cb64db0cdca22e
    7b9cd822586e781e6f82d84f347f37d8f37d1ba6aadcf46f418a1308e714c1aeb58cb6e36f1dec39d71fb70d34ebd00
    d455b18aec831fc68e3cb004cd6fdc1dc53c97836f8afb7e903db02ddc5eed15b32de60f4ea3b90ae11bc4cd3b133b7
    c

Final ciphertext (base64):
13KO6gHix7OCe+wtrNToCch8/6BrTH35mWv6LIdl2ETTlb26bGr1mys14X3FXroQwwgu1SVhXfmGQIt1lwbNS9AFkHs4
    +djvtlXFdMN5kDDEQPgMeejrqakaukG2SfAdxUDcvG52XebCf4oIzVz9W1xmrMps75hevXy4Tatq
    /BKXMrqYZKbiiM4stk2wzcoi57nNgiWG54Hm+C2E8OfzfY83Obpqrc9G9BihMI5xTBrrWMtuNvHew51x
    +3DTTrOA1FWxiuyDH8aOPLAEzW/cHcU8l4Nvivt+kD2wLdxe7RWzLeYPTqO5CuEbxMO7Ezt8

--- Decryption Process ---

Decrypted text:
Georgia is a small country with a great and ancient history. For centuries, it has been a constant
    battleground for great empires due to its strategic location. However, it still managed to
    survive and maintain its statehood.
```

Result 2.

ISSN 1512-1232

Conclusion

The main feature of the proposed encryption system is its speed. The algorithm processes data with O(n) time complexity. This ensures efficient encryption and decryption. XOR-based encryption is inherently fast because it requires minimal computational resources, and the use of genetic operations introduces randomness without significantly increasing the time. Although the system is fast, from a security perspective, it does not reduce the level of protection. Encryption rounds ensure that the plaintext remains hidden even if the initial encryption fails to hide it sufficiently. However, security still depends on the uniqueness of the key and the reliability of the PRNG. Although the system is already optimized for fast performance, its parallelization and hardware acceleration can further improve its efficiency without compromising security.

References

[1] Kochladze Z, Beselia L., Benidze N., Creating an encryption block algorithm using prng and genetic operations. Of the XII international scientific-practical conference internet-Education-science ies-2020. Ukraine,Vinnytsia,VNTU,2020.pp5-6.

[2] Spillman R.,Janssen M., Nelson B., Kepner N.,, Use of Genetic Algorithm in Cryptanalysis of Simple Substituion Cipher, Cryptologia, 1993.Vol.17, No.4, pp. 367-377.

[3] Kochladze Z., Beselia L., Cracking of the Merkle–Hellman Cryptosystem Using Genetic Algorithm, Transections on Sciense and Tecnology , Volume 3, No. 1-2: Science and Natural Resources , 201, pp. 291-296 .

[4] Garg P., Genetic algorithm Attack on Simplified Data Encryption Standard algorithm, International journal Research in Computing Science, ISSN1870-4069, 2006. Pp23-28.

[5] Gorodilov A., Morozenko B., genetic algorithm for finding the key's length and cryptanalysis of the permutation cipher, International Journal "Information Theories & Applications" Vol.15 , 2008. Pp.175-260.

Article received: 2025-03-03